Approved for public release; distribution is unlimited.

INCREMENTAL ON-LINE TYPE INFERENCE

by

Thomas Lewis Robinson
Lieutenant, United States Navy
B.A., University of Colorado, Boulder, 1987

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March, 1994

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Computer Science Department Naval Postgraduate School 6b. OFFICE SYMBOL (*if applicable*) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943
NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*if applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NC

TITLE (*Include Security Classification*)

INCREMENTAL ON-LINE TYPE INFERENCE

PERSONAL AUTHOR(S)
Robinson, Thomas Lewis

a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Master's Thesis	FROM 06/93 TO 03/94 .	March 1994	67

SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the ficial policy or position of the Department of Defense or the United States Government.

COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Software, On-line Type Inference, Interactive Programming Environment

ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Type inference in *interactive* programming environments falls short in two respects. The ability to typ eck definitions one at a time, and to type check some definitions but not all after one definition is modifie called *incremental on-line type inference*. Current *interactive* programming environments perform *batch* typ ference and require extensive type recomputation for small changes.

We give an algorithm for on-line type inference that is implemented as an attribute grammar. From thi ammar an editor was automatically generated that performs on-line type inference.

The editor infers types incrementally due to a well-known reduction we used from Hindley-Milner type infe ıce to first-order unification. Unlike other efforts, our algorithm for on-line type inference is truly incrementa

DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
2a. NAME OF RESPONSIBLE INDIVIDUAL Dennis M. Volpano	22b. TELEPHONE (*Include Area Code*) (408) 656–3091 22c. OFFICE SYMBOL CSVo

FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PA
UNCLASSIFIED

ABSTRACT

Type inference in *interactive* programming environments falls short in two respects. The ability to type check definitions one at a time, and to type check some definitions but not all after one definition is modified is called *incremental on-line type inference*. Current *interactive* programming environments perform *batch* type inference and require extensive type recomputation for small changes.

We give an algorithm for on-line type inference that is implemented as an attribute grammar. From this grammar an editor was automatically generated that performs on-line type inference.

The editor infers types incrementally due to a well-known reduction we used from Hindley-Milner type inference to first-order unification. Unlike other efforts, our algorithm for on-line type inference is truly incremental.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank Dr. Dennis Volpano for his infinite patience and guidance. His assistance, encouragement, and insight both academically and personally gave me a perspective, I would not otherwise have gleaned, and am eternally grateful for.

I would also like to thank Dr. Timothy Shimeall for his help in reviewing this thesis.

I would like to express my gratitude to my family for their support. To my wife, Sally; for her encouragement, industrious nature, and peaceful spirit. To my children; for reminding me of the most important issues of life.

Finally, I would like to thank Jonathan Hartman, wherever he may be, for his superbly documented LaTeX style files and examples that made the idiosyncrasies of the Naval Postgraduate School thesis format requirements tolerable.

I. INTRODUCTION

Interactive programming environments can significantly improve resource utilization and programmer productivity. Current interactive programming environments, however, have not completely resolved some critical issues and consequently typically offer limited support for the programmer. Programming environments of the future will need to offer much more in the way of language-based support, such as type checking. A language-based editor ought to detect and report type errors as they occur. The traditional edit-compile-debug-run environment can require hundreds or even thousands of lines of code to be recompiled for something as trivial as a missing semicolon. The problem is that the environment in which the program is created typically knows very little about the language being used. It is only when the program is passed to the compiler that the programmer receives any feedback.

An interactive programming environment integrates many of the separate aspects of current programming environments. Type checking in an interactive programming environment is *on-line* in the sense that definitions may be type checked one at a time as they are entered into the system. On-line type checking will provide the programmer immediate feedback. This means that at any stage of program development the interactive environment will provide feedback about the type correctness of the program even if the program is only partially complete. Consider the following partial definition for a function that computes the length of a list given in Standard ML [Ref. 11] notation:

```
fun length l =
  case l of
    [] => <exp>
    | <pat> => <exp>
;
```

In the notation for Standard ML "[]" is the empty list. The definition for *length* is not complete (the incomplete term for a pattern is denoted by <pat> and the unspecified expressions are denoted by <exp>), yet we may still derive a type according to the rules discussed in Chapter II and given in [Ref. 8]. The type we can derive is $\forall\alpha.\forall\beta.list\,\alpha \rightarrow \beta$. This notation is the standard notation found in [Ref. 8] and is discussed in more detail in Chapter II.

Interactive programming environments exist that perform limited type checking but they are not on-line. For a pair or set of mutually recursive definitions, the programmer must explicitly provide all the definitions of the mutually recursive set, otherwise an error results. This is because environments such as Standard ML use an extension of Damas and Milner's algorithm *W* [Ref. 8], that is an algorithm for the batch type checking problem. A batch type checker reports an error upon detecting an unbound identifier. A unbound identifier is one for which no definition is provided. The problem is that all the definitions in the sequence must be supplied otherwise the batch type checker complains about the unbound identifiers.

Type checking in an *interactive* programming environment is an *on-line* problem vice a batch problem. Definitions are type checked one at a time and the type checker must not object to undefined free identifiers. Rather, when a definition is provided the type checker must ensure that the definition is used correctly. Consider again the definition for *length* given above. We can fill in some of the missing information.

```
fun length l =
  case l of
    [] => 0
    | <pat> => <exp>
;
```

Now we can type the definition of *length* $\forall\alpha.list\,\alpha \rightarrow int$. As we shall see later, this type implies *length* will accept a list of elements of any type as input and return the

length of the list. This type can be determined even though the definition is not complete.

We complete the definition for *length* so that we have:

```
fun length l =
  case l of
    [] => 0
    | (h::t) => 1 + (length t)
;
```

In Standard ML the notation (h::t) means the concatenation of the head and tail of a list. Completing the definition, however, does not provide any more type information for *length* and we are left with $\forall\alpha.list\,\alpha \rightarrow int$ for the principal type of *length*.

We propose an extension of W for *on-line* type checking. (In this thesis type checking and type inference are used synonymously.) The model we use is a consistently-attributed parse tree specified by an attribute-grammar. Implicit in the model is incremental recomputation of types. In addition, recomputing a type can be reduced to reunification using a well-known reduction from Hindley-Milner style type checking to first-order unification. Unlike other extensions of W [Ref. 13], for on-line type checking, our on-line type checker is truly incremental.

II. THE HINDLEY-MILNER TYPE SYSTEM

In this chapter we review the history and basis for our type system. In Section A we describe the simply-typed lambda calculus. In Section B we describe the Hindley-Milner type system and the grammar for the expressions typed in their type system. We will also describe parametric polymorphism and look at an example that shows how polymorphic functions are useful. In Section C we look at the type inference rules of the Damas and Milner type system. In Section D we review Damas and Milner's algorithm W which infers a type for an expression of the grammar given in Section B. This chapter puts the thesis contribution in the context of other work, namely the simply-typed lambda calculus, the Hindley-Milner type system, and the polymorphic lambda calculus.

A. THE SIMPLY-TYPED LAMBDA CALCULUS

The simply-typed lambda calculus has types: $\tau ::= \rho \mid \tau_1 \rightarrow \tau_2$. Small greek letters are used to represent type variables (type variables have values that range over all named and anonymous types definable in a language.) We will use the standard notation $x : \sigma$ to mean that the expression x has type σ. In this type system we can infer the types of lambda expressions such as: $\lambda x.x : \alpha \rightarrow \alpha$. We can also type more complex expressions such as: $(\lambda x.x)\, \lambda x.x : \beta \rightarrow \beta$.

In the simply-typed lambda calculus, however, we would be unable to type the expression, $(\lambda y.y\ y)\, \lambda x.x$. The simply-typed lambda calculus fails to infer a type for this expression because λ in the simply-typed lambda calculus exhibits monomorphic abstraction. This means that each instance of y in $(y\ y)$ must have the same type. In order to type $(\lambda y.y\ \ y)\, \lambda x.x$, we must be able to instantiate a unique type for

each instance of y in $(y\ y)$. The simply-typed lambda calculus, however, fails in this regard. But the expression $(\lambda y.y\ y)\,\lambda x.x$ reduces to the equivalent expression $(\lambda x.x)\,\lambda x.x$. We have already seen that this expression can be given the type $\beta \rightarrow \beta$. So we have two expressions that are equivalent but only one may be typed in the simply-typed lambda calculus.

B. HINDLEY-MILNER TYPE SYSTEM

The Hindley-Milner type system extends the types of the simply-typed lambda calculus with type schemes (quantified expressions of type variables), so now in addition to τ types we also have type schemes. The complete type system is:

$$\begin{array}{lcl} \tau & ::= & \rho \mid \tau_1 \rightarrow \tau_2 \\ \sigma & ::= & \forall\alpha.\sigma \mid \tau \end{array}$$

The Hindley-Milner type system is able to infer a more general type than the simply-typed lambda calculus. In the Hindley-Milner type system, for example, we can infer a more general type for the identity function: $\lambda x.x : \forall\alpha.\alpha \rightarrow \alpha$.

The Hindley-Milner type system, however, still fails to infer a type for the expression $(\lambda y.y\ y)\,\lambda x.x$, because of the limitation of monomorphic lambda abstraction. But the Hindley-Milner type system provides the *let* expression, that allows the equivalent expression: **let** $y = \lambda x.x$ **in** $y\ y$ **ni**, to be correctly typed. *Let* in the Hindley-Milner type system exhibits polymorphism and, using their type system, we can infer the correct type: $\forall\alpha.\alpha \rightarrow \alpha$, for the expression **let** $y = \lambda x.x$ **in** $y\ y$ **ni**. This is better than the simply-typed lambda calculus in which we could not type the expression $(\lambda y.y\ y)\lambda x.x$ at all.

In order to type the expression $(\lambda y.y\ y)\,\lambda x.x$ we need a polymorphic lambda calculus, which is beyond the scope of this thesis. A polymorphic lambda calculus is a second order calculus that would allow a type for each instance of y in $(y\ y)$ to be instantiated uniquely.

$$
\begin{array}{rcl}
e & \rightarrow & x \\
 & | & e\ e' \\
 & | & \lambda x.e \\
 & | & \textbf{let } x = e \textbf{ in } e' \textbf{ ni}
\end{array}
$$

Figure 2.1: Expressions in the Hindley-Milner type system.

The Hindley-Milner type system will infer a type for a single expression. The grammar for expressions in the Hindley-Milner type system, shown in Figure 2.1, is the lambda calculus with the addition of *let* expressions. Some authors include an additional production to the ones given in Figure 2.1, $e \rightarrow c$, where c is a constant such as *true*, *false*, *1*, *2*, ... This production is just a special case of $e \rightarrow x$ and we will use just the production $e \rightarrow x$ where x can be an identifier or a constant such as *true*, *false*, *1*, *2*, ...

Lambda abstraction $\lambda x.x$ is the identity function. By definition, this function applied to any argument simply returns the argument. So that if we applied $\lambda x.x$ to *true* we would get *true*, and if we applied $\lambda x.x$ to 1 we would get 1. In the Hindley-Milner type system we are able to infer a much more general type for this expression. We can now say, $\lambda x.x$ has type $\forall\alpha.\alpha \rightarrow \alpha$, which is more general than $\beta \rightarrow \beta$ in the sense that α can be instantiated to any type.

We can also define more complicated expressions built up from basic expressions. For instance, we can define a function *first* $= \lambda x.\lambda y.x$, which will take as input a pair and return the first element of the pair, so if it were applied to (1 2) it would return 1. Actually the lambda calculus is curried, which means functions can only be applied to a single argument. So the definition we have given for *first* actually takes as input the first element of a pair and returns a function that takes as input the second element of a pair. For this function, the value returned would be the first element of the two input elements.

Similarly we could define a function that returns the second element of a pair: $\lambda x.\lambda y.y$, which will take as input a pair and return the second element of the pair, so if it were applied to (1 2) it would return 2.

A conditional expression can similarly be defined:

$$Cond = \lambda x.\lambda y.\lambda z.z\ x\ y,$$

and if we also define *True* which is defined the same as first,

$$True = \lambda x.\lambda y.x,$$

then we could apply

$$Cond\ 1\ 2\ True,$$

where we mean:

$$((((\lambda x.\lambda y.\lambda z.z\ x\ y)\ 1)\ 2)\ True).$$

The parentheses have been added to emphasize the fact that the lambda calculus is curried and the λ abstraction can only be applied to one argument. The order in which this may be reduced is:

$$(((\lambda y.\lambda z.z\ 1\ y)\ 2)\ True),$$

where the bound occurrence of x has been replaced by 1. Next we may perform the following reduction:

$$((\lambda z.z\ 1\ 2)\ True),$$

where y has been replaced by 2. Finally we can reduce the expression to:

$$((True\ 1)\ 2),$$

where z has been replaced by *True*. This can also be reduced as follows:

$$\begin{array}{lll} ((True\ 1)\ 2) & & \\ = & & (((\lambda x.\lambda y.x)\ 1)\ 2) \\ \rightarrow & & ((\lambda y.1)\ 2) \\ \rightarrow & & 1 \end{array}$$

This reduction gives us the expected result. Recall the original expression was *Cond 1 2 True*. If this were expressed as the equivalent *If true then 1 else 2 fi* we would see the expected result immediately. Since *true* is always *true* we get 1 from the *then* branch of the conditional expression just as we would expect.

Polymorphism in the Hindley-Milner type system is called parametric polymorphism. Functions that operate on parameters of different types are polymorphic. Suppose that we could define a function `length` that returns the length of a list of any type. We can say `length` has type $\forall\alpha.list\,\alpha \rightarrow int$ adn therefore is polymorphic. It returns the length of any list. [Ref. 1: pg. 364]

Aho, Sethi, and Ullman claim Ada is polymorphic, albeit a restricted polymorphism. It is worth taking a look at an example of what might be considered a polymorphic Ada module. Their claim is that *generics* in Ada are polymorphic. In this context *generic* is an Ada reserved word. It is true that an Ada *generic* module may be compiled without complete type specifications. In order to use such a function, however, the *generic* module must be instantiated at run-time. The *generic* module must be instantiated for each type of list that uses the function `length`. Thus at run-time, an instance of the *generic* module will exist for every type for which the module has been instantiated and each module will be monomorphic. [Ref. 1: pg. 364]

Let's look at an Ada specification for a *generic* function `length`, that will return the length of a linked-list of any type.

```
generic
  type list is private;
  with function Next (e: list) return list;
  with function Empty (e: list) return boolean;
function length (l: list) return positive;
```

We must also provide a body for the function.

```
function length (l: list) return positive is
  len: positive := 0;
  lptr : list := l;
begin
  while not Empty(lptr) loop
    len := len + 1;
    lptr := Next(lptr);
  end loop;
  return len;
end length;
```

This comprises a complete compilation unit in Ada. A bit more work is required to use the function `length` in a program. If we look at the specification for the *generic* function we see that three parameters are required. First, we must know how to access the elements of the list so we need a pointer to a list element which should contain a field with a pointer to another list element and to be useful at least one data field. The second and third parameters are functions which are required to manipulate the type of list we have specified in the first parameter. Since this is a *generic* module we have to explicitly provide these functions when the *generic* function is instantiated for a specific list type. For each type of list that uses the *generic* function `length`, a separate instantiation is required. Each instantiation requires separate code for the specific list type.

At compile time the *generic* function `length` could be given type $\forall\alpha.\mathit{list}\,\alpha \rightarrow \mathit{int}$ that leads one to believe *generics* in Ada may be polymorphic. At run-time, however, each instance of `length` can only have type $\mathit{list}\,\tau \rightarrow \mathit{int}$ for some particular τ. The fact that at run-time there exists an instance of the function `length` for every type for which the *generic* function `length` has been instantiated means that *generics* in Ada are not polymorphic, and in fact Ada is monomorphic.

In Standard ML a function that finds the length of a list can be written that is polymorphic.

```
fun length l =
  case l of
    [] => 0
    | (h::t) => 1 + (length t)
;
```

Length in Standard ML could also be typed $\forall\alpha.list\,\alpha \rightarrow int$ and this would hold true at run-time as well [Ref. 1: pg. 365].

In the Hindley-Milner grammar the *let* expression provides *let* polymorphism. If e is polymorphic then in the expression **let** $x = e$ **in** e' **ni**, every free occurrence of x in e' can be assigned an instance of the type of e. This implies that each occurrence of x in e' may have a different type depending on how each occurrence of x is used in e'.

Different strategies have been proposed for typing *let* expressions. Damas and Milner's algorithm W handles *let* expressions by typing the definition of the *let*-bound *id* and then typing the body of the *let* expression in an environment that includes a generic type for the *let*-bound *id*. Here we are using the term generic to refer to a type that may be instantiated and not a *generic* unit in Ada. See [Ref. 4] for more about generic types.

Another strategy is to replace every occurrence of the *let*-bound *id* in the body of the *let* expression with the definition of the *let*-bound *id*. This strategy has a disadvantage. If the *let*-bound *id* does not occur in the body of the *let* expression then the replacement will not occur. Consequently during type analysis the definition of the *let*-bound *id* will never be processed. This would allow type errors in the body of the *let*-bound *id's* definition to go undetected. This is unacceptable under strict semantics for *let* expressions.

TAUT: $A \vdash x : \sigma$ $\quad x : \sigma$ in A

INST: $\dfrac{A \vdash e : \sigma}{A \vdash e : \sigma'}$ $\quad \sigma > \sigma'$

GEN: $\dfrac{A \vdash e : \sigma}{A \vdash e : \forall\alpha\sigma}$ $\quad \alpha$ not free in A

COMB: $\dfrac{A \vdash e : \tau' \to \tau \quad A \vdash e' : \tau}{A \vdash (e\ e') : \tau}$

ABS: $\dfrac{A_x \bigcup \{x : \tau'\} \vdash e : \tau}{A \vdash (\lambda x.e) : \tau' \to \tau}$

LET: $\dfrac{A \vdash e : \sigma \quad A_x \bigcup \{x : \sigma\} \vdash e' : \tau}{A \vdash (\textbf{let } x = e \textbf{ in } e' \textbf{ni}) : \tau}$

Figure 2.2: The Rules of the Hindley-Milner type system.

C. THE RULES OF DAMAS AND MILNER

The notation used in this paper is that used in [Ref. 8]. Briefly, A is a set of assumptions for which the following holds true: A contains at most one assumption about each identifier x, and A_x is the set of assumptions with no assumption for x. We use the notation $x : \sigma$ to mean that x has the type σ. Also we use the following notation to mean, from the set of assumptions A we can infer type σ, where σ is a type scheme, for an expression e: $A \vdash e : \sigma$. Also, for α and β, type variables and τ a type scheme: $[\beta_i/\alpha_i]\tau$ means to replace all free occurrences of the α_i's in τ with the β_i's. Damas and Milner provided the rules shown in Figure 2.2 for type inference [Ref. 8].

D. ALGORITHM *W*

Damas and Milner gave the following algorithm for typing a single expression of the grammar given in Figure 2.1 with respect to an initial assumption set A. W returns a substitution S and a type variable τ [Ref. 8].

$W(A, e) = (S, \tau)$ where

1. If e is x and there is an assumption $x : \forall\alpha_1 \ldots \alpha_n\tau'$ in A then $S = Id$ and $\tau = [\beta_i/\alpha_i]\tau'$ where the β_i's are new.

2. If e is $(e_1\ e_2)$ then let $W(A, e_1) = (S_1, \tau_1)$ and $W(S_1A, e_2) = (S_2, \tau_2)$ and $U(S_2\tau_1, \tau_2 \rightarrow \beta) = V$ where β is new; then $S = VS_2S_1$ and $\tau = V\beta$.

3. If e is $\lambda x.e_1$ then let β be a new type variable and $W(A_x \bigcup x : \beta, e_1) = (S_1, \tau_1)$; then $S = S_1$ and $\tau = S_1\beta \rightarrow \tau_1$.

4. If e is **let** $x = e_1$ **in** e_2 **ni**, then let $W(A, e_1) = (S_1, \tau_1)$ and $W(A, e_1) = (S_1, \tau_1)$ and $W(S_1A_x \bigcup \{x : \overline{S_1A}(\tau_1)\}, e_2) = (S_2, \tau_2)$; then $S = S_2S_1$ and $\tau = \tau_2$.

The substitution $S = [\beta_i/\alpha_i]$ returned by W must be applied to the type variable τ, and the resulting τ type must closed with respect to the initial set of assumptions A. The result is a type scheme for the expression.

W infers the type for a single expression. There is no implicit mechanism to deal with top-level definitions. Any extension of W to a "program" must be done explicitly.

1. Most General Unifier

Damas and Milner's algorithm W uses Robinson's unification algorithm. The unification algorithm U has the following property: If $U(\tau, \tau')$ returns V, then V unifies τ and τ', i.e., $V\tau = V\tau'$. Also, if S unifies τ and τ', then $U(\tau, \tau')$ returns

some V and there is another substitution R such that $S = RV$. V involves only variables in τ and τ' [Ref. 8: pg. 210].

The substitution V that we are interested in, is the *most general unifier*, that imposes the fewest constraints on the variables in the expression [Ref. 1: pg. 370].

2. Principle Type Scheme

W infers a principal type scheme for an expression. This implies that any other type scheme for the expression is a generic instance of the principal type scheme. This is dependent on the unification algorithm returning the *most general unifier* [Ref. 8: pg. 208].

III. ON-LINE TYPE INFERENCE WITH UPDATE

In this chapter we will look at the two key elements of a type checker for an interactive programming environment. First, we look at what makes the type checker on-line. Second, we look at the effect of an update which affords us an opportunity to perform type checking incrementally.

Interactive programming environments would provide much greater leverage to the programmer if the environment had the ability to perform type checking as definitions are input to the system, one definition at a time, rather than waiting until a series of definitions is complete and then analyzing them for type errors. Additionally, given a sequence of definitions that have already been built up in an interactive environment, a small change to one definition should not cause the entire sequence to be retyped unnecessarily. If the system is well designed it should be able to use as much information from previous typings and limit recomputation.

There are two issues involved. The first is what we refer to as on-line type inference. Secondly, we are concerned with the semantics of what an update to a sequence of definitions means for the entire sequence.

Consider defining a function g that uses a function f such that g is free in E. If we encode this in Standard ML using *let* as

```
let val f = fn ... in
  let val g = fn ...f... in E
  end
end
```

then we must pay attention to the dependency. The definitions of f and g are sequenced. However, Standard ML does provide simultaneous declarations so we could write it as a simultaneous declaration where g is defined first:

```
fun g() = ...f()...
and f() = ...;
E
```

But now suppose h is also free in the definition of g. If we don't include a definition for h as well in the simultaneous declaration, then the Standard ML interpreter complains that h is an unbound variable or constructor when in fact the interpreter could very well infer a type for f.

The problem is that simultaneous declarations in Standard ML and *letrec* definitions in Scheme are typed in batch mode. That is, all definitions must be supplied before a type is inferred for any one of the defined functions. Clearly this is undesirable in a programming environment where definitions are typically given in any order and the need for type analysis begins before all definitions of objects comprising a system or even a subsystem are present. The problem at hand then is what we call *on-line* type inference. We are proposing a programming environment in which definitions can be made in any order and are continuously type-checked.

A. ON-LINE TYPE INFERENCE

Type checking definitions, one at a time, is called on-line type checking. If a definition, g has been provided and we have inferred a type for g, then when we infer a type for f in which g occurs free, we will use an instantiation of the type of g to infer a type for f. However, if no definition has been provided for g and we are inferring a type for f in which g occurs free then we can give g an instantiation of $\forall\alpha.\alpha$ as necessary in f. This assumption can take place without restricting the type of f or g, nor any loss of generality. This implies that meaningful type analysis may

take place at any time, and not just when a program or sequence of definitions is complete. This also means that if g occurs more than once in f, each occurrence may be typed uniquely because the generic type $\forall\alpha.\alpha$ can be instantiated as necessary depending on how g is used in f.

We propose an environment that will infer a type for a definition that may contain free identifiers and may possibly be incomplete. Those free identifiers may be defined and thus may have an associated type. In the case of a partial definition missing elements will be represented by a place holder. Thus the expression will be complete in the context where place holders are considered valid terms. If a definition f contains free identifiers whose types are not known or is incomplete, we may still be able to infer a type for f. We will assume that an undefined identifier has the most general type $\forall\alpha.\alpha$. This can be done without loss of generality, restricting the type of any free identifier nor imposing any constraints on the type of a term containing place holders. If f contains a free identifier g that has not been defined, a type may still be inferred for f. If later a definition is provided for g, then the type of f is updated reflecting any changes imposed by the type of g.

We could extend the grammar given in Chapter II for expressions to include place holders:

$$
\begin{array}{rcl}
e & \rightarrow & < expression > \\
 & | & x \\
 & | & e\ e' \\
 & | & \lambda x.e \\
 & | & \textbf{let } x = e \textbf{ in } e' \textbf{ ni}
\end{array}
$$

The production $e \rightarrow < expression >$ denotes a place holder for an undefined or partial expression. This change appears minor, but if we allow such expressions in the grammar and are still able to infer a type for the expression then we can infer types for partial definitions.

Recursive definitions are handled using the fixed point combinator Y [Ref. 2: pg. 164]. The fixed point combinator Y has the property that for an expression M,

$YM = M(YM)$. This allows recursive and mutually recursive definitions to be typed, but forces the user to use the fixed point combinator Y.

To demonstrate the handling of mutual recursion, consider defining the functions even and odd. Using the grammar for expressions given above where we use the notation *if* $z\ x\ y$ for the conditional, we first define a function *pair*:

$$pair = \lambda x.\lambda y.\lambda z.z\ x\ y$$

Next we define functions *first* and *second*:

$$\begin{aligned} first &= \lambda x.\lambda y.x \\ second &= \lambda x.\lambda y.y \end{aligned}$$

Now we must provide a definition for the fixed point combinator Y:

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Y has the property that (YM) for some function M is $M\,(YM)$. Finally, we must define a tuple for *even* and *odd*:

$$\begin{aligned} P = (Y(\lambda x.((&pair \\ &(\lambda n.(\text{if } (= n\ 0)\ true\ ((x\ second)(-n\ 1))))) \\ &(\lambda n.(\text{if } (= n\ 0)\ false\ ((x\ first)(-n\ 1))))))) \end{aligned}$$

Now we can define *even* as P applied to *first* and *odd* as P applied to *second*.

$$\begin{aligned} even &= (P\ first) \\ odd &= (P\ second) \end{aligned}$$

Let's look at an example: First, let's rewrite P to make it easier to read:

$$P = (YM)$$

where

$$\begin{aligned} M = (\lambda x.((&pair \\ &(\lambda n.(\text{if } (= n\ 0)\ true\ ((x\ second)(-n\ 1))))) \\ &(\lambda n.(\text{if } (= n\ 0)\ false\ ((x\ first)(-n\ 1)))))) \end{aligned}$$

Now we can apply *even* to a number:

$$\begin{aligned} &(\text{even } 2) \\ &\rightarrow ((P\ first)\ 2) \\ &\rightarrow (((YM)\ first)\ 2) \\ &\rightarrow (((M\ (YM))\ first)\ 2) \\ &\rightarrow ((((\lambda x.((pair \\ &\qquad (\lambda n.(\text{if } (= n\ 0)\ true\ ((x\ second)(-n\ 1))))) \\ &\qquad (\lambda n.(\text{if } (= n\ 0)\ false\ ((x\ first)(-n\ 1)))))) \\ &\qquad (YM))\ first)\ 2) \\ &\rightarrow ((((pair \\ &\qquad (\lambda n.(\text{if } (= n\ 0)\ true\ (((YM)\ second)(-n\ 1))))) \\ &\qquad (\lambda n.(\text{if } (= n\ 0)\ false\ (((YM)\ first)(-n\ 1)))))) \\ &\qquad first)\ 2) \\ &\rightarrow ((\lambda n.(\text{if } (= n\ 0)\ true\ (((YM)\ second)(-n\ 1))))\ 2) \\ &\rightarrow (((YM)\ second)\ 1) \\ &\rightarrow (((M(YM))\ second)\ 1) \\ &\rightarrow ((((\lambda x.((pair \\ &\qquad (\lambda n.(\text{if } (= n\ 0)\ true\ ((x\ second)(-n\ 1))))) \\ &\qquad (\lambda n.(\text{if } (= n\ 0)\ false\ ((x\ first)(-n\ 1)))))) \end{aligned}$$

$$
\begin{aligned}
&\quad (YM))\ second)\ 1) \\
&\rightarrow ((((pair \\
&\qquad (\lambda n.(\text{if } (= n\ 0)\ true\ (((YM)\ second)(-n\ 1))))) \\
&\qquad (\lambda n.(\text{if } (= n\ 0)\ false\ (((YM)\ first)(-n\ 1)))))) \\
&\qquad second)\ 1) \\
&\rightarrow ((\lambda n.(\text{if } (= n\ 0)\ false\ (((YM)\ first)(-n\ 1))))1) \\
&\rightarrow (((YM)\ first)\ 0) \\
&\rightarrow (((M\ (YM))\ first)\ 0) \\
&\rightarrow ((((\lambda x.((pair \\
&\qquad (\lambda n.(\text{if } (= n\ 0)\ true\ ((x\ second)(-n\ 1))))) \\
&\qquad (\lambda n.(\text{if } (= n\ 0)\ false\ ((x\ first)(-n\ 1)))))) \\
&\qquad (YM))\ first)\ 0) \\
&\rightarrow ((((pair \\
&\qquad (\lambda n.(\text{if } (= n\ 0)\ true\ (((YM)\ second)(-n\ 1))))) \\
&\qquad (\lambda n.(\text{if } (= n\ 0)\ false\ (((YM)\ first)(-n\ 1)))))) \\
&\qquad first)\ 0) \\
&\rightarrow ((\lambda n.(\text{if } (= n\ 0)\ true\ (((YM)\ second)(-n\ 1))))\ 0) \\
&\rightarrow (true)
\end{aligned}
$$

As expected, *even* applied to 2 returns *true*.

In our type system the definitions we type are partially ordered. Ordered definitions and mutual recursion would normally cause a problem. We have a solution, however, which is to use the fixed point combinator Y and tuples. The brief example we just looked at demonstrates how a system can implement mutually recursive definitions with a tuple and the fixed point combinator Y. Since we can define mutually recursive definitions using a tuple and Y, the definitions given in a program are partially ordered.

In an off-line or batch type system, no definition may contain undefined free identifiers, otherwise an error results. Standard ML is an off-line type inference system. In our system, definitions may have undefined free identifiers and the order those definitions are entered is irrelevant. Additionally a definition may not even be complete but contain place holders for expressions and the type inference system will still return a valid type if one exists. Suppose we entered the following definition:

$$\text{foo} = \lambda < \text{identifier} > . < \text{exp} >.$$

The place holders for an identifier and expression indicate we have not completely specified the definition of *foo*, however, we can still infer a type for *foo*: $\forall\alpha.\forall\beta.(\alpha \rightarrow \beta)$. Type analysis takes place with no restrictions imposed by the place holders and we can infer the most general type for the expression.

Damas and Milner's algorithm W infers a type for a single expression. We would like to have the ability to type a series of named expressions or a "program." A named expression is simply a definition. Typing expressions one at a time allows programs to be develop in an incremental top-down fashion. This is not so easy to do in batch systems since a free identifier for which no definition has been provided causes an error. A batch system forces bottom-up development where all definitions must be defined before being referenced. For example, defining g and then f in terms of g should be the same as defining f in terms of g and then defining g. In an imperative language such as Ada, if we define f in terms of g but have not already defined g, nor provided a specification for g, during compilation we will get an error. Ada requires that either g be defined or a specification be provided prior to its use. Either way, this forces the programmer to worry about how function definitions are ordered when in fact ordering functions at the same nesting depth is irrelevant to the meaning of the program.

B. UPDATE

In an interactive programming environment there is a unique problem when we have a sequence of definitions and we *update* or modify one of the definitions. The effects of an *update* varies among languages. There are at least four possible interpretations for updating a sequence of definitions [Ref. 9].

1. An update to a definition supersedes the original definition and is used in all subsequent references to the definition but does not change definitions prior to the new (updated) definition. This is ML's interpretation.

2. An update to a definition is an augment to a partially defined definition and is incorporated in all subsequent references to the definition but does not change definitions prior to the new (updated) definition.

3. An update to a definition supersedes the original definition and all references both prior and subsequent to the new (updated) definition are changed. This is the behavior we desire.

4. An update to a definition augments the original definition and all references to the definition both prior and subsequent to the new (updated) definition are changed. This is Prolog's interpretation.

Standard ML's interpretation requires that the user re-enter any definition that depends on the modified definition if the effect of the modification should be propagated. For example, in Standard ML if we have a definition f and then a definition g which references f, and then update the definition of f so now we have f', the reference to f in g is not changed, and consequently g is not changed either. This is not desirable because if there are many other definitions besides g which reference f and we want them all to refer to the new definition f', then every definition in which

f occurs free that we want to reference the new definition f' must be re-entered after the new definition for f'. A better solution would be to have an environment that allows a user "to go back" and modify the original definition. In this sense a syntax-directed editor provides the logical framework in which to allow updates to be made directly to the original text. Update will now replace the original definition and all references to the definition will refer to the new definition.

Typically for environments like Prolog, Scheme, and Standard ML a file is loaded and the entire file is interpreted and any type errors are reported at that time. If a definition is to be changed then the file is edited, loaded, and processed. So even if the change was small and, only a few definitions needed to be processed, the rest of the definitions in the file are processed unnecessarily. The problem is how to handle small changes to a large set of definitions with the minimum amount of work [Ref. 10].

There are two issues that determine the incremental aspects of an update. First, if we can preserve information when we type a definition we can reduce the amount of work that is necessary to recompute the type of that definition. Assume we have a definition f in which g occurs free. A type may already have been inferred for f when a modification to g necessitates that f have its type reinferred. If we use as much information as possible from the earlier typing of f when its type is reinferred then we can type f incrementally.

Second, if the dependencies are carefully observed only those definitions that depend on a modified definition or whose type changes as a result of a modification, must be retyped. It is possible to partially order the definitions because we have the fixed point combinator Y to handle recursive and mutually recursive definitions. We can represent the dependencies as a pair (f, g), that means f depends on g. If we have a sequence of definitions with the following dependencies, $\{(f, g), (f, h), (h, i)\}$ and f were modified, only f would need to have its type inferred. If i were modified

then types would need to be inferred for f, h, and i, but not g. So, by observing the dependencies we can limit the amount of work needed to infer types for the sequence of definitions.

Since we have an environment that allows the programmer to edit the original definition there is no question what a reference to a modified definition should be. The original definition no longer exists so any reference must be to the new *updated* definition. This is even more important in an on-line environment where definitions may be entered in any order. In f a reference to g may refer to a definition either above or below f in the parse tree. The ordering does not matter in the on-line environment. So if f references g and g is modified yielding g' the reference in f is clearly to g'.

IV. ON-LINE TYPE INFERENCE THROUGH ATTRIBUTION

In this chapter we look at a special class of grammars called attribute grammars and provide an attributed grammar for on-line type inference. The attribution is circular but nonetheless is an on-line type inference algorithm. In Section A we look at attribute grammars in general and provide the semantics for the attribute equations of an attributed grammar. In Section B we will look at a circular set of attribute equations for on-line type inference and discuss the meaning of such a set of attribute equations.

A. ATTRIBUTE GRAMMARS

A context-free grammar (CFG) is a tuple, $G = (N, \Sigma, S, P)$ where N is a set of symbols, Σ is a set of terminals, $S \in N$ is the start symbol, and P is a set of productions. This formalization can be easily extended to allow the symbols in $(N \bigcup \Sigma)$ to have values. This extension is a special class of grammars called Attribute Grammars (AGs). The names of values associated with a symbol in a grammar are called attributes. The attributes for a grammar's symbols can be divided into two categories: inherited and synthesized. Inherited attributes can be calculated from parent's and sibling's attributes. Synthesized attributes can be calculated from children's attributes and other attributes at the same node. The equations from which an attribute's value is determined are called attribute equations. [Ref. 1: pg. 280]

Formally, an attribute grammar can be expressed as a 5-tuple [Ref. 7]:

$$AG = (G,\ A,\ VAL,\ SD,\ SC).$$

G is a context-free grammar that we just discussed. A is a set of attributes such that each attribute $a \in A$ ranges over a domain of values denoted by $Dom(a)$. VAL is the set of values that an attribute a can assume,

$$VAL = \{Dom(a) \mid a \in A\}.$$

Each attribute is associated with a symbol of $(N \bigcup \Sigma)$. If X is a symbol of G then the attributes of X are denoted by:

$$A_x = \{X.a \mid a \in A\}.$$

For each occurrence X_i of a symbol X in a production p there is an attribute instance $X_i.a$ for all $a \in A_x$. For each production p, the set of all attribute instances is:

$$A_p = \{X_i.a \mid a \in A_x \text{ and } X_i \in p\}.$$

A is partitioned into two disjoint sets AI and AS, the inherited and synthesized attributes respectively. The inherited attributes of the start symbol must be null, $AI_S = \emptyset$. The synthesized attributes of the terminal symbols must also be null, $AS_X = \emptyset$ if $X \in \Sigma$.

SD is the set of semantic definitions for the productions of P.

$$SD = \{SD_p \mid p \in P\}.$$

A semantic definition defines the value of an attribute instance in A_p. The value depends only on other attribute instances in A_p. There can be only one such semantic definition that assigns a value to an attribute a in A_p. Given a semantic definition $f : Dom(b_0) \times \cdots \times Dom(b_k) \rightarrow Dom(a)$, then $(X_i.a = f(X_0.b_0, \ldots, X_k.b_k)) \in SD_p$. Thus, semantic definitions are local to a particular production $p \in P$.

SC is the set of semantic conditions (predicates). There is one semantic condition for each production $p \in P$.

$$SC_p \in Dom(b_0) \times \cdots \times Dom(b_k) \rightarrow BOOL,\ b_j \in A_p.$$

$$
\begin{array}{rcl}
P = \quad E & \rightarrow & E + F \\
 & | & E - F \\
 & | & F \\
F & \rightarrow & i
\end{array}
$$

Figure 4.1: Productions for the calculator grammar.

Production	Semantic Rules
$E \rightarrow E + F$	$E_1.v = E_2.v + F.v$
$E \rightarrow E - F$	$E_1.v = E_2.v - F.v$
$E \rightarrow F$	$E.v = F.v$
$F \rightarrow i$	$F.v = i.lexval$

Figure 4.2: Semantic definitions for the productions in Figure 4.1.

A sentence S in $L(G)$ is in $L(AG)$ iff for each use of production p in a derivation of S, the values of its attribute instances satisfy SC_p.

Consider the following grammar G, where $N = \{E, F\}$, $\Sigma = i$. $S = E$, and the set of productions P is given in Figure 4.1. The grammar defines a calculator for the $+$ and $-$ operators. This CFG can be attributed so that we have an attributed grammar AG. First, we let the set of attributes for E and F be $\{v\}$, so $A = \{v\}$. The domain for v is the natural numbers so $VAL = Z$. The semantic definitions are given in Figure 4.2. In Figure 4.2 the notation E_1 refers to the first or leftmost occurrence of the symbol E in the production. Multiple occurrences are counted left to right. For example, $E_1.v$ refers to the attribute named v of the first symbol E in a production. If a term E occurs only once in a production its attributes are simply denoted $E.a$.

The parse tree for this grammar can then be decorated with the value of the expressions such that at each node we store the value of the subtree rooted at that node. For example, the decorated parse tree for the expression $3 + 7$ is given in

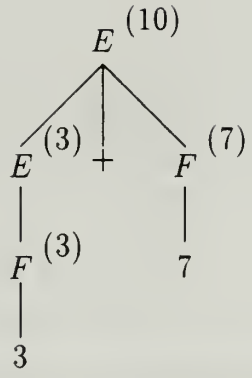

Figure 4.3: Attributed parse tree for the expression $3 + 7$.

Figure 4.3. [Ref. 1: pg. 280]

B. A CIRCULAR ATTRIBUTION FOR ON-LINE TYPE INFERENCE

A special case of an AG is a circular AG (CAG). Non-circular AGs (NCAGs) have been found useful in a variety of applications. CAGs, however, have generally not been considered useful, and in fact were considered ill-formed and meaningless until recently [Ref. 1: pg. 334] and [Ref. 12]. Semantic equations, for a circular attribution, that employ monotonic operators (over some complete partial order), define a unique greatest (least) fixed point that may be interpreted as the meaning of the circular attribution [Ref. 12]. Sagiv et al. also give an algorithm for converting these CAGs to NCAGs [Ref. 12].

On-line type inference can be characterized as computing the least fixed point of a set of circular attribute equations. We will provide a CAG that is an on-line type inference algorithm. The CFG for the language used in our type inference system is similar to the grammar used in Hindley-Milner style type system.

A set of attribute equations is circular if there is a mutual dependency between two attributes. When an attribute $S.i$ depends on an attribute $S.a$ and $S.a$ depends

$$
\begin{array}{lcl}
T & \rightarrow & P \\
P & \rightarrow & D\ P \mid D \\
D & \rightarrow & Id = M \\
M & \rightarrow & Id \\
 & \mid & M\ N \\
 & \mid & \lambda Id.N \\
 & \mid & \textbf{let } Id = M \textbf{ in } N \textbf{ ni}
\end{array}
$$

Figure 4.4: CFG for the on-line type inference system.

Production	Semantic Definitions
$T \rightarrow P$	$P.inhTE = \emptyset$
$P \rightarrow D\ P$	$P_1.synTE = D.synTE \bigcup P_2.synTE$
	$P_2.inhTE = D.synTE$
	$D.inhTE = P_1.inhTE \bigcup P_2.synTE$
$P \rightarrow D$	$D.inhTE = P.inhTE$
	$P.synTE = D.synTE$
$D \rightarrow Id = M$	$D.synTE = \{Id,\ TypeOf(M,\ D.inhTE)\} \bigcup D.inhTE$

Figure 4.5: Semantic definitions for the grammar in Figure 4.4.

on $S.i$, where i is an inherited attribute and a is synthesized, the AG is circular.

The CFG we will use for the on-line type inference problem is given in Figure 4.4. The set of attributes for this grammar is $A = \{inhTE, synTE\}$. The domain of the attributes in A are type environments ordered by subset inclusion. The semantic definitions for the CAG are given in Figure 4.5. There are no semantic conditions for this CAG, so $SC = \emptyset$.

The inherited and synthesized type environment attributes in A denoted by $inhTE$ and $synTE$ respectively are used to pass the inferred type environment up and down the tree. A type environment maps id's to type schemes. At the node for each definition a type will be inferred and then added to the type environments being passed up the tree and down the tree. The attribute equations for the pro-

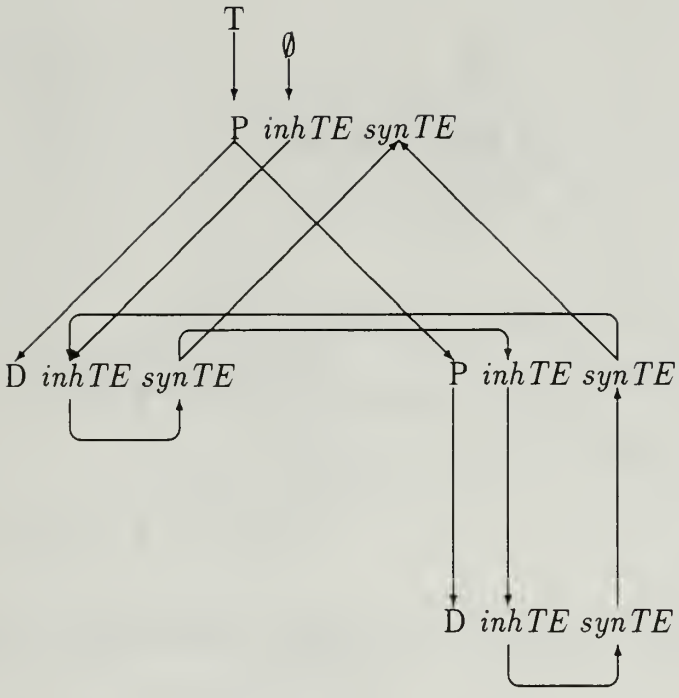

Figure 4.6: A circular attribution for on-line type inference.

duction $P \rightarrow D\ P$ add the type of a definition to the inherited and synthesized type environment attributes for the symbol P.

The type environment that is inherited at a node D in the production $P \rightarrow D\ P$, consists of the types of all the definitions both above and below the node in the parse tree. These types are propagated up the tree in *P's* synthesized type environment attribute and down the tree in *P's* inherited type environment attribute.

In the attribute equation for the production $D \rightarrow Id = M$, the function *TypeOf* returns the principal type of the expression M with respect to the inherited type environment $D.inhTE$. The circularity of this attribution is shown in Figure 4.6.

A trivial program in this language might look like:

$$a = \lambda x.x;$$
$$b = a;$$

The parse tree for this program is shown in Figure 4.7. The attribute equations for

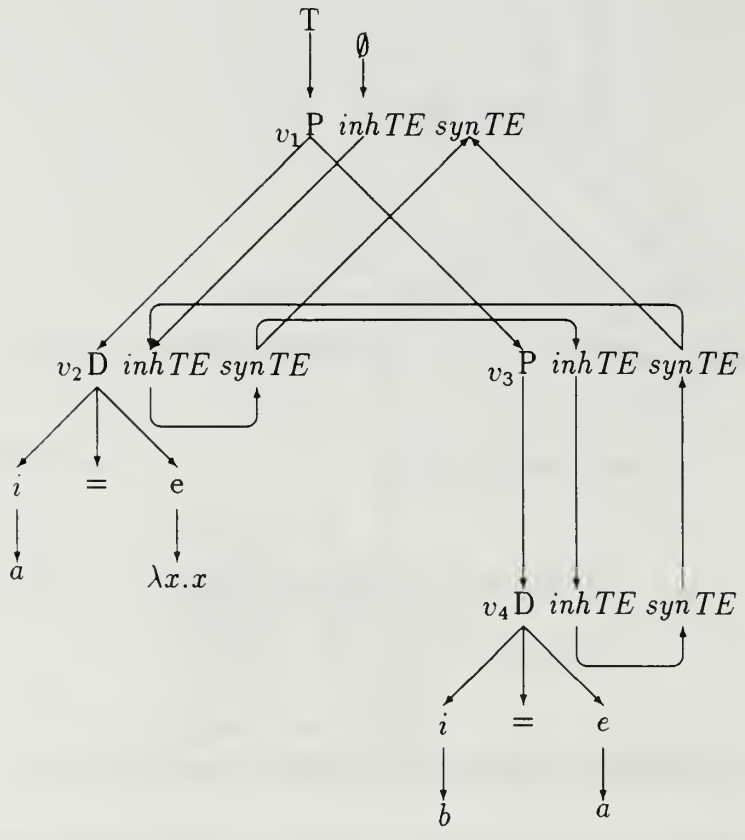

Figure 4.7: The parse tree for $a = \lambda x.x$ and $b = a$.

the program shown in Figure 4.7 yield the system of equations in Figure 4.8. The least fixed point of a circular set of attribute equations such as the ones in Figure 4.8 is a solution to the on-line type inference problem.

A CAG is meaningful if a few conditions are met. By Tarski's theorem if a system S is such that all the functions f_x are monotonic in all their arguments, and all functions are defined over complete partial orders (CPOs) then a fixed point exists [Ref. 12: pg. 38]. Unlike Sagiv et al. we're interested in the least fixed point.

The system of equations in Figure 4.8 can be shown to meet Tarski's conditions, as discussed in Sagiv's et al.'s work, and thus has a least fixed point, which is a type environment providing types for all definitions whose types can be determined in the input stream. The right hand sides of the equations in Figure 4.8 can be viewed

$$
\begin{aligned}
v_1.inhTE &= \emptyset && (4.1)\\
v_1.synTE &= v_2.synTE \bigcup v_3.synTE && (4.2)\\
v_2.inhTE &= v_1.inhTE \bigcup v_3.synTE && (4.3)\\
v_2.synTE &= (a, \alpha \rightarrow \alpha) \bigcup v_2.inhTE && (4.4)\\
v_3.inhTE &= v_2.synTE && (4.5)\\
v_3.synTE &= v_4.synTE && (4.6)\\
v_4.inhTE &= v_3.inhTE && (4.7)\\
v_4.synTE &= \text{if } (a, \tau) \in v_4.inhTE \; rmthen && (4.8)\\
&\qquad (b, TypeOf(a, A, v_4.inhTE)) \bigcup v_4.inhTE\\
&\quad \text{else } v_4.inhTE\\
v_5.inhTE &= v_4.synTE && (4.9)
\end{aligned}
$$

Figure 4.8: Semantic equations for program in Figure 4.7

as functions that compute the value of the attribute given in the left hand side of the equations. Equation 4.1 is trivially monotonic because $\emptyset$ is a constant function. Equations 4.2–4.4 are monotonic since, $\forall x.\forall y.\forall z.\ x \subseteq y \Rightarrow x \bigcup z \subseteq y \bigcup z$ therefore the union operator $\bigcup$, is monotonic in both its arguments. Equations 4.5–4.7 and 4.9 are implicitly defined using identity which is monotonic. To show Equation 4.8, denoted $f_{4.8}$, is monotonic we must show that $\forall x.\forall y.\ x \subseteq y \Rightarrow f_{4.8}(x) \subseteq f_{4.8}(y)$

Suppose $x \subseteq y$. Then

Case I. If $(a, \tau) \in x \Rightarrow f_{4.8}(x) = x \bigcup \{(b, \tau)\}$, since $x \subseteq y$ then $(a, \tau) \in y \Rightarrow f_{4.8}(y) = y \bigcup \{(b, \tau)\}$

Case II. If $(a, \tau) \notin x \Rightarrow f_{4.8}(x) = x$. If $(a, \tau) \in y$ then $f_{4.8}(y) = y \bigcup \{(b, \tau)\}$, and if $(a, \tau) \notin y$ then $f_{4.8}(y) = y$. In either of these two cases, $f_{4.8}(x) \subseteq f_{4.8}(y)$.

The domain of the equations in Figure 4.8 is the power set of the set of all typed definitions. If our program consists of definitions for a and b, then the powerset of typed definitions consists of $\{\emptyset, \{(a, \tau_a)\}, \{(b, \tau_b)\}, \{(a, \tau_a), (b, \tau_b)\}\}$ which can be ordered by set inclusion. The resulting ordering is a CPO shown in Figure 4.9.

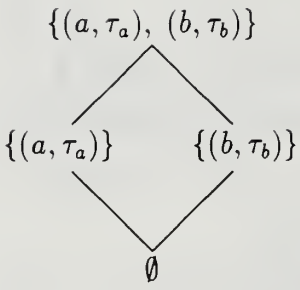

Figure 4.9: Type environments ordered by set inclusion.

We are concerned with the least fixed point. If $\{(a, \alpha \rightarrow \alpha),\ (b, \alpha \rightarrow \alpha)\}$ is a solution to the type inference problem for the program $a = \lambda x.x;\ b = a;$ then $\{(a, \alpha \rightarrow \alpha),\ (b, \alpha \rightarrow \alpha),\ (c, \alpha \rightarrow \alpha)\}$ is also a solution, meaning it is also a fixed point but it is not the least fixed point. The set $\{(a, \alpha \rightarrow \alpha),\ (b, \alpha \rightarrow \alpha)\}$ is the least fixed point for the semantic definitions of the program in Figure 4.7.

While we know that the circular attribution given in Figure 4.5 has meaning and can be shown to have a least fixed point which can be taken to be a solution to the type inference problem, tools exist for NCAGs. Sagiv et al. have a technique for transforming CAGs to NCAGs [Ref. 12]. This transformation from a circular attribution to a non-circular attribution is being investigated.

Another, alternative would be to use Farrow's evaluator generator which is capable of accepting a circular attribution and generating a fixed-point-finding attribute evaluator. [Ref. 5]

V. AN INCREMENTAL ALGORITHM FOR ON-LINE TYPE INFERENCE

In this chapter we present an incremental algorithm for on-line type inference that is an incremental type-checker. It is a syntax directed editor, but can be thought of as a programming environment. As such, it provides the programmer an environment to evaluate the types for definitions of the form **val** $i = e$, where **val** is a keyword to denote a definition. Identifiers i, denote the name of the definition. The expressions, denoted by e, have the form we saw in Chapter IV. The definitions all have top-level scope much like definitions in Standard ML. Unlike Standard ML, however, the scope of the definitions in our programming environment is the entire program and not only the rest of the program that appears after the definition. Moreover, the order definitions are entered is irrelevant.

First, we look at an incremental algorithm for on-line type inference. It relies on a reduction from the Hindley-Milner type system to first-order unification. The remainder of the chapter is devoted to an example that demonstrates the reduction, unification and resulting type computations.

A. THE INCREMENTAL ALGORITHM

The algorithm is incremental in two respects. First, if the type of a definition must be reinferred as much work as possible from the previous typing is used in the retyping. Second, only those definitions whose types may have changed will have types reinferred. There are three events that may cause a definition's type to change. First, when a definition is input to the system. Second, when a definition is modified.

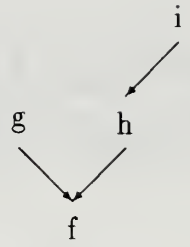

Figure 5.1: Dependency partial order.

Third, when a definition's type changes as a result of a change to the type of another definition on which it depends.

If a definition f must be retyped as a result of a new type being generated for a definition g, that occurs free in f, we can retype f incrementally if we can use some information from the original computation for the type of f. If we observe the dependencies we can be even more incremental. For example, if we have definitions f, g, h, i, and dependencies $\{(f,g),(f,h),(h,i)\}$, shown in Figure 5.1, where (f,g) means f depends on g, and then modify the definition of i, we must infer a new type for i, since it was modified. In addition, if the type of i changes, we will have to infer new types for those definitions that depend on i. But we have to be careful about the order in which the definitions are typed after i's type is reinferred. If we modify the initial set of definitions such that i is free in both h and j, then we have the dependencies shown in Figure 5.2. Both h and j depend on i, but j also depends on h. If i is modified then we may need to infer types for f, h, and j, but we must type h before f and j because of the dependencies (j,h) and (f,h). If we recompute h's type and it does not change then only j remains to be typed, not f, even though f transitively depends on i, which was modified.

Before we give the incremental algorithm we must first define some notation. Let A be a set of type assumptions mapping the given operators to type schemes.

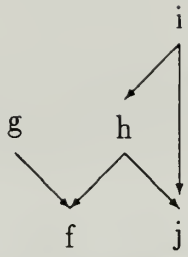

Figure 5.2: Direct and transitive dependency for (j, i).

The operators in the initial assumption set are *cons, hd, tl, nil,* and Y. Let G (a DAG) be the dependency graph for the definitions in the program. Let eqn_M and α_M denote the type equations and the type variable respectively, for a term M. Construction of eqn_M and α_M is discussed in Section B. We also let $TE[v' : \sigma]$ mean update type environment TE, so that v' now has type σ. The algorithm uses function *TypeOf* defined by

$TypeOf(M, TE)$ = **let** $S = Unify(eqn_M, TE)$ **in**

$close(S\alpha_M, TE)$.

Unify needs a type environment since, as we shall see, eqn_M may contain references to types of other top-level definitions. *TypeOf* applies the most general unifier of eqn_M, to α_M and then closes the resulting type giving the principal type for M.

The incremental algorithm is defined below:

Input: A DAG G of definitions, a type environment TE and a vertex v of G.
Output: A type environment.
begin
  $Affected = \{v\}$
  while $Affected \neq \emptyset$ **do**
    delete the least element $v' \in Affected$
    let $\sigma = TypeOf(M, TE)$ where v' is defined as term M **in**
      if $\sigma \neq \sigma'$ where $v' : \sigma' \in TE$ **then**
        $Affected = Affected \bigcup \{x \mid (x, v') \in edges(G)\}$

```
                TE = TE[v' : σ]
            fi
        ni
    od
return TE
end
```

Let R be the dependency relation for a set of definitions S. We must compute the transitive closure of the dependency relation R. The transitive closure is anti-symmetric by virtue of the way mutual recursion is handled using the fixed point combinator Y. Therefore the transitive closure is antisymmetric and transitive, and thus a partial order say P. We can extend P to a consistent total (linear) order L. We write the dependency (x, m) as $m \leq x$. For a modified definition m let *Affected*(m) be the set of affected definitions. *Affected*$(m) \subseteq S$, is defined as $\{x \mid (x, m) \in P\}$. So *Affected*(m) is the set of all definitions that depend on the modified definition m. The *Affected* set is totally ordered by L, so we can recompute the types of each definition in *Affected* in the order given by L.

We can take the example from Figure 5.2 and again consider what happens if i is modified. A total (linear) order for this set of definitions is $L = (g, i, h, j, f)$. When the incremental algorithm is called, *Affected* is set to i. Since i is the only element in *Affected* it is the least element and is removed from *Affected*. The type of i is computed and we will assume the type has changed from the previous type of i. *Affected* is updated to include h and j. The type environment TE is also updated with the new type for i. On the next iteration, *Affected* is $\{h, j\}$ and since $h \leq j$, h is the next definition to be retyped. If the type of h has changed, then f would be added to the *Affected* set, otherwise only j's type remains to be computed. This process continues until *Affected* is empty.

$E(M, A) =$

case M **of**

$\lambda x.M$: **let** $A' = A \cup \{x \mapsto \beta\}$ β new var **in**
let $(\alpha, e) = E(M, A')$ **in**
$(\gamma, \{e\} \cup \{\gamma = \beta \rightarrow \alpha\})$ where γ is new

$(M\ N)$: **let** $(\alpha, e_1) = E(M, A)$ **in**
let $(\beta, e_2) = E(N, A)$ **in**
$(\gamma, \{e_1\} \cup \{e_2\} \cup \{\alpha = \beta \rightarrow \gamma\})$ γ new

x: **if** $x : \forall \bar{\alpha}.\tau \in A$ **then**
let $\tau' = [\beta_i/\alpha_i]\tau$ where β_i's are new **in**
$(\gamma, \{\tau' = \gamma\})$ where γ is new
else
$(\beta, \{t_x = \beta\})$ β new

Figure 5.3: Algorithm E, to generate the type equations for an expression.

B. REDUCTION

Note that *TypeOf* uses α_M and eqn_M produced for a term M. The latter is a set of type equations corresponding to an instance of first-order unification. This instance is obtained by a well-known reduction from type inference to unification [Ref. 14]. The α_M is a type variable to which the most general unifier can be applied to get a principal type for M. The reduction is achieved by algorithm E of Figure 5.3. It takes a term M and an assumption set A as input and returns a pair (α_M, eqn_M), where α_M is a type variable and eqn_M is an associated set of type equations for M. The type obtained by closing the application of eqn_M's most general unifier to α_M is the principal type of M.

The notation t_x means the type of the definition for x. In E we generate type equations where t_x stands for the type of x. When we unify the equations for a definition say y in which x occurs free we must instantiate the type of x and replace **each** occurrence of t_x in the type equations for y with an instantiation of the type

of x. Since the definitions will be partially ordered we know we will have already typed the definition for x prior to typing the definition for y if x occurs free in y, thus we will be able to instantiate the type of x for each occurrence of t_x in the type equations of any definition y, each occurrence getting a new instantiation.

The correctness of algorithm E can be stated as follows: Suppose $E(M, A) = (\alpha_M, eqn_M)$, where M is a let-free expression, A is a set of type assumptions, α_M is a type variable, and eqn_M is a set of type equations. Then S is a unifier of eqn_M iff $A \vdash M : S\alpha_M$.

1. Algorithm L – Lifting Let Expressions

To build the type equations necessary for unification we must first deal with *let* expressions. Algorithm E does not handle *let* expressions. The system proposed by Wand and O'Keefe also ignores *let* expressions [Ref. 14]. Cardelli's system handles *let* expressions in the same fashion as Damas and Milner [Ref. 4] [Ref. 8]. Their strategy, however, is not suitable for an on-line environment. Typing the *let*-bound *id's* definition and then typing the body of the *let* expression will not work in an environment with top-level definitions, because a *let*-bound *id's* definition may contain free identifiers. To meet the requirements for being on-line, a *let*-bound *id's* definition would have to be partially ordered, along with the rest of the top level definitions and typed accordingly.

In order to handle *let* expressions we need a different strategy. To construct the equations necessary for unification, we must lift the *let* expressions from the terms of our programs so that all our expressions are *let*-free.

There are two considerations that determine how *let* expressions must be handled. First, we require *let* expressions to exhibit polymorphism. This requirement implies that each occurrence of a *let*-bound *id* in the body of the let expression is not restricted to the same type. Each occurrence of a *let*-bound *id* may be used

$$\textbf{val } a = \textbf{let } x = y \textbf{ in } x\ 2 \textbf{ ni}$$
$$\textbf{val } y = \lambda z.z$$

Figure 5.4: Free identifier in a *let*-bound definition.

uniquely, and thus may have a unique type associated with it.

Second, we must allow for top-level naming that implies a definition may contain free identifiers. In the Hindley-Milner type system a type is inferred for a single expression that may also contain free identifiers, but those free identifiers may not refer to another top-level expression. A limited form of naming is permitted in the Hindly-Milner type system using *let* expressions. Thus in our type system where a free identifier may occur anywhere in the expression including a *let*-bound *id's* definition, we have to treat *let* expressions differently.

For example, consider the trivial program fragment in Figure 5.4. There is a dependency in a on y. We only consider the free identifiers in a when we determine the dependencies. The *let*-bound *id* x is not free in a. But, we must consider **any** free identifier in a even in the definition of the *let*-bound identifier. If we were to infer a type for the *let*-bound *id* x and then type the body of the *let* expression using the reduction technique we are proposing then we would have to apply the results of typing x to the typing of the body of the *let* expression.

The solution, when we have an expression **let** $x = e$ **in** e' **ni**, is to replace each occurrence of x in e' with e. This allows us to maintain *let* polymorphism and deal with top-level naming. If a *let*-bound *id's* definition contains free identifiers, the substitution of e for x in e' will introduce a free identifier in the body of the *let* expression, which we know how to handle, and eliminate the necessity of applying the results of one typing to another.

This strategy has the disadvantage that if the *let*-bound *id* does not occur in the body of the *let* expression then the definition of the *let*-bound *id* will not be

$L(exp, env) =$

case exp **of**

x:	$(x, exp') \in env \ ? \ exp' : exp$
$\lambda x.M$:	$\lambda x.L(M)$
$(M\ N)$:	$(L(M)\ L(N))$
let $x = M$ **in** N **ni**:	$L(N,\ env[x \mapsto L(M, env)])$

end case

Figure 5.5: Algorithm L, lifts the let expressions from the definitions.

typed. We could get around this limitation by forcing the equations for the definition of the *let*-bound *id* to be included with the equations for the *let* expression's body and unified regardless whether *id* is referenced.

We have an algorithm L shown in Figure 5.5 which simply replaces each occurrence of a *let*-bound *id* in the body of the *let* expression with the *let*-bound *id's* definition. L takes as input an expression and an environment which consists of pairs of *let*-bound identifiers and their definitions and returns a *let*-free expression. The environment is initially empty. The algorithm recursively calls itself until all the terminal nodes have been reached. When a *let* expression is encountered a binding is added to the environment and the body of the *let* expression is processed using the new environment.

2. EXAMPLE REDUCTION AND UNIFICATION

In this section we present an example that demonstrates the type checker in action. We look at the type equations that are generated by E and the final types for the definitions. The final types are inferred by unifying the equations returned by E, applying the substitution that is returned by unification, and closing the type.

Suppose we enter the following definition:

$$g = \lambda y.\lambda z.cond\ z\ nil\ (f\ y)$$

In g *cond* and f are free, and no definition is provided for them yet. Also free in g is *nil* which is defined in the initial assumption set and has type $\forall\alpha.\mathit{list}\,\alpha$. E will return an associated type variable and a set of type equations. In these examples we will show the sub-expression from which the type equation was generated next to the type equation. For g, the type variable returned by E is ξ, and the type equations that get generated by E are:

Subexpression	Type equation	
$cond$	$t_{cond} = \iota$	(5.1)
z	$\beta = \theta$	(5.2)
$cond\ z$	$\iota = \theta \rightarrow \kappa$	(5.3)
nil	$\mathit{list}\,\eta = \zeta$	(5.4)
$(cond\ z)\ nil$	$\kappa = \zeta \rightarrow \lambda$	(5.5)
f	$t_f = \delta$	(5.6)
y	$\alpha = \gamma$	(5.7)
$f\ y$	$\delta = \gamma \rightarrow \epsilon$	(5.8)
$((cond\ z)\ nil)\ (f\ y)$	$\lambda = \epsilon \rightarrow \mu$	(5.9)
$\lambda z.((cond\ z)\ nil)\ (f\ y)$	$\nu = \beta \rightarrow \mu$	(5.10)
$\lambda y.\lambda z.((cond\ z)\ nil)\ (f\ y)$	$\xi = \alpha \rightarrow \nu$	(5.11)

Unification of the type equations yields a substitution that when applied to ξ and then closed gives us the type $\forall\alpha.\forall\beta.\forall\gamma.(\alpha \rightarrow (\beta \rightarrow \gamma))$ for g.

If we continue and provide a definition for $cond = \lambda x.\lambda y.\lambda z.z\ x\ y$, E will return the type variable β and the following type equations for *cond*:

Subexpression	Type equation
z	$\epsilon = \eta$

x	$\alpha = \theta$
$z\ x$	$\eta = \theta \rightarrow \iota$
y	$\gamma = \kappa$
$(z\ x)\ y$	$\iota = \kappa \rightarrow \lambda$
$\lambda z.(z\ x)\ y$	$\zeta = \epsilon \rightarrow \zeta$
$\lambda y.\lambda z.(z\ x)\ y$	$\delta = \gamma \rightarrow \zeta$
$\lambda x.\lambda y.\lambda z.(z\ x)\ y$	$\beta = \alpha \rightarrow \delta$

The type of *cond* is $\forall\alpha.\forall\beta.\forall\gamma.(\alpha \rightarrow (\beta \rightarrow ((\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow \gamma)))$. Now we can retype g since we now have a new type for *cond*. All that is required, is to reunify the type equations given above for g, only this time we will be able to instantiate the type for *cond*, and not have to use an instantiation of the assumption $\forall\alpha.\alpha$, for the type of *cond*. This, however, does not change the type of g.

Suppose we define $f = \lambda x.x$. This definition will cause g's type to change. The type variable for f returned by E is γ and its type equations are:

Subexpression	Type equation
x	$\alpha = \beta$
$\lambda x.x$	$\gamma = \alpha \rightarrow \beta$

The type of g is now $\forall\alpha.\forall\beta.\forall\gamma((\alpha \rightarrow (\mathit{list}\ \beta \rightarrow \gamma)) \rightarrow (\alpha \rightarrow \gamma))$.

But if we change the definition of f to $f = \lambda x.\mathit{nil}$, we will get a type error for g. The type variable for f returned by E is γ and its new type equations are:

Subexpression	Type equation
nil	$\mathit{list}\ \alpha = \beta$
$\lambda x.\mathit{nil}$	$\gamma = \delta \rightarrow \beta$

This changes the type of f to $\forall\alpha.\forall\beta.\alpha \rightarrow \mathit{list}\ \beta$, that causes unification to fail on g's type equations. When g's type equations are reunified Equation 5.1 gives us $\iota = (\pi \rightarrow (\rho \rightarrow ((\pi \rightarrow (\rho \rightarrow \sigma)) \rightarrow \sigma)))$. From Equation 5.3, $\kappa = (\rho \rightarrow ((\pi \rightarrow (\rho \rightarrow \sigma)) \rightarrow \sigma))$. Equations 5.4 and 5.5 give us $\lambda = ((\pi \rightarrow (\mathit{list}\ \eta \rightarrow \sigma)) \rightarrow \sigma)$. In Equation 5.6 we must instantiate the type of f. This will give us $\delta = \tau \rightarrow \mathit{list}\ \phi$. Then from Equation 5.8 we find $\epsilon = \mathit{list}\ \phi$. Finally, when we try to unify Equation 5.9 we find we have to unify $\mathit{list}\ \phi$ with $(\pi \rightarrow (\mathit{list}\ \eta \rightarrow \sigma))$ which fails. As a result–g exhibits a type error.

C. THE SYNTHESIZER GENERATOR

We have implemented an on-line type checker using a tool called The Synthesizer Generator (SynGen) [Ref. 6]. This tool was chosen for its ability to rapidly produced a syntax directed editor with an X-Windows interface. The language used in SynGen is the Synthesizer Specification Language (SSL).

There are three main parts of a specification in SSL. First, is the abstract syntax for the underlying language of the editor. The second part is the attribution and the accompanying attribute equations. Lastly, are the unparsing rules which determine how the program's parse tree is to be traversed. Part of the unparsing rules also determine what is displayed at each node and what the user is allowed to edit. The unparsing rules also control how expressions and terms are input to the system.

Unfortunately, SSL does not allow circular attribute equations. In order to get around the limitations of SSL we chose to perform the type inference at the root of the parse tree. This is a change to the circular attribution given in Chapter IV. The abstract syntax used in our implementation looks quite similar to that given in Figure 4.4. A program is simply a list of definitions or bindings of identifiers and expressions of the lambda calculus (with the addition of *let* expressions).

An attribution is provided in our implementation that is used to pass up the tree the type variables and type equations generated by E for each definition. The attribution also passes a type environment back down the tree. At the node for each definition the type variables and type equations are generated by E and passed up the tree. At the root, types are inferred for the definitions and a type environment is passed back down the tree. Then at the node for each definition a lookup is performed on the environment that is being passed down the tree and the type for the current node is pulled from the environment and displayed.

There is another significant optimization we would like to be able to make, but is not possible in the framework of SSL. Given a sequence of definitions $f, g, h, \ldots$, and a set of dependencies $(f, g), (g, h), (h, i), \ldots$, and h is updated, we would like to be able to infer a type for only h and those definitions that transitively depend on h, in this case g and f. Unfortunately SSL does not provide any mechanism to detect whether a parse tree has been changed, so we end up reunifying the type equations for all definitions. This is undesirable but the implementation is still incremental to the extent the type equations are not regenerated. This is strictly a limitation of SSL.

To demonstrate the interface generated by *SynGen* a brief example is shown in Figures 5.6 through 5.18. Figure 5.6 shows the programming environment at start up. The top pane is a message pane. The middle pane is the editing window for the defined language. The bottom pane is a context aid that shows at what node of the parse tree the cursor is currently resting. The context pane also shows some available transformations of the current node which would take the user one level deeper in the parse tree. The displayed transformations are up to the editor designer so additional transformations may be defined but not displayed.

Figure 5.7 shows the editor after the context has been changed to **def list**. A program consists of a set of assumptions and a list of definitions. The user-input

assumption set may be null but there will be a place holder shown in the editor window.

In Figure 5.8 the cursor has been placed on the place holder for a definition. As you can see the context has changed to a **def list**. When the context selection **def** in Figure 5.8 is made the definition place holder is transformed to the place holders for an identifier and expression as is shown in Figure 5.9. The keyword **val** and the symbol "=" are also inserted.

The definition has been named g and the context has been moved to **expression** in Figure 5.10. Once the definition is named the type is displayed. Note that g is given the universal type $\forall\alpha.\alpha$. The reason no type is provided until the definition is named is due to the fact that the display routine does a lookup based on the definitions name in the type environment. The available transformations for an expression are shown in the context pane at the bottom of the window. The expression can also be just an identifier although this transformation is not explicitly listed. The displayed transformations are up to the editor designer. Optional input modes permit a very flexible combination of explicit and implicit input modes that make the editor as rigid or as flexible as the designer wishes. One extreme would be to force the user to explicitly enter all input with a mouse making selections from the context pane while the opposite extreme allows a syntactically correct expression to be entered from the keyboard. This allows for a friendly interface that does not require explicit mouse selection for every program transformation.

Figure 5.11 shows the editor after the expression has been transformed to a λ abstraction. Figure 5.12 shows an identifier entered for the λ abstraction and the context advanced to the expression.

In Figure 5.13 a second lambda abstraction has been entered. Figure 5.14 shows the identifier for the second lambda abstraction and the context advanced. Figure 5.15 shows the rest of the definition for g. The free identifiers in g are *cond*,

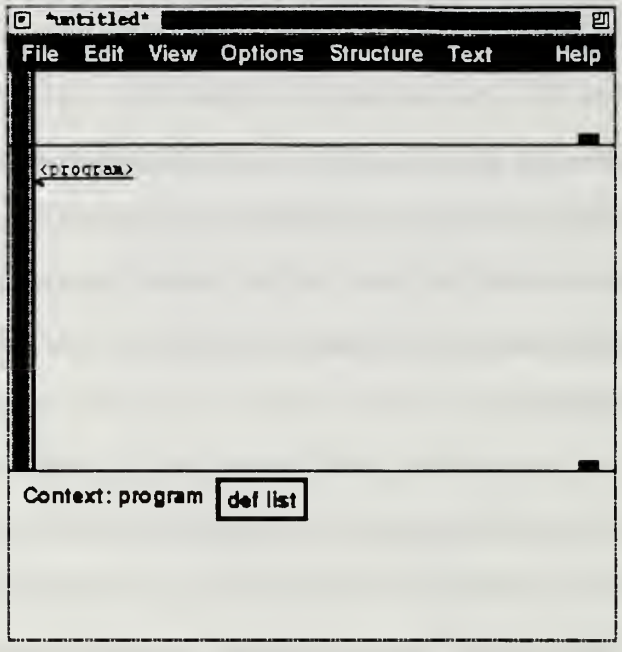

Figure 5.6: The initial view of the syntax directed editor.

nil, and f. Recall *nil* is one of the built in operators with type $\forall\alpha.list\,\alpha$.

In Figure 5.16 the definition for *cond* has been entered. Providing a definition for *cond* causes g's type equations to be reunified but the type of g is not changed.

In Figure 5.17 f has been defined as the identity function. This change causes g's type equations to once again be unified and a new type for g is inferred. This time the type of g does change to reflect the constraints imposed by the type of f and *nil*.

The final display in Figure 5.18 shows the effects of redefining f such that it causes g's type equations to be reunified. This time, however, the unification process fails due to the inconsistent use of f in g and the type system returns a type error for g.

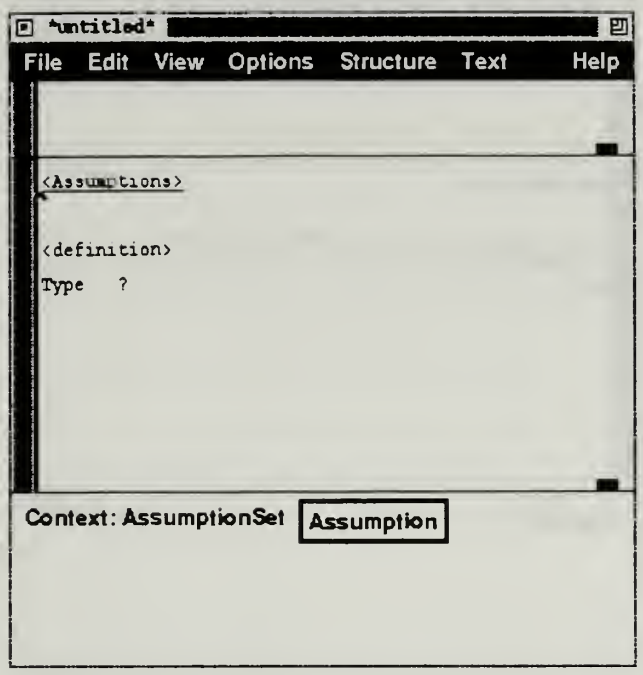

Figure 5.7: The view when the context **def list** is selected.

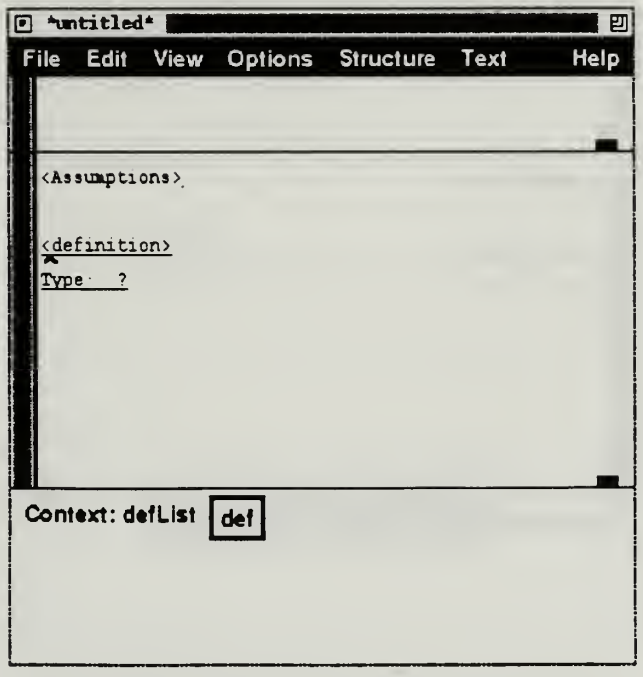

Figure 5.8: The view when the cursor is placed in the first definition.

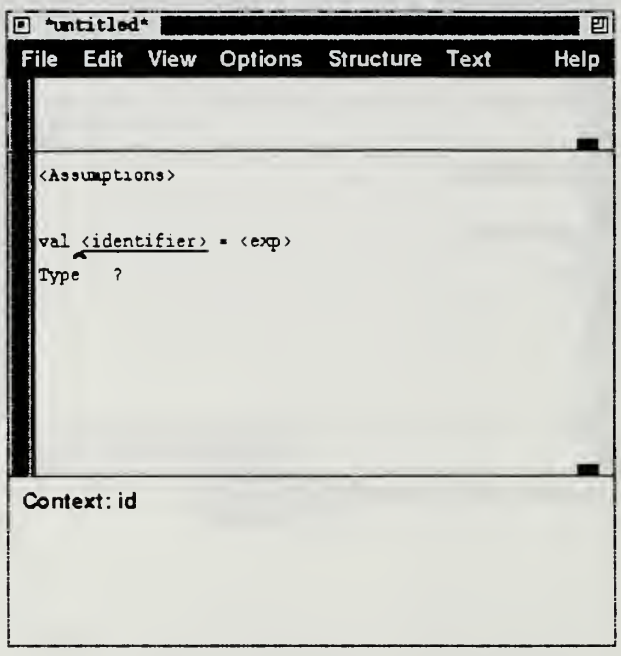

Figure 5.9: The view after the context **def** is selected.

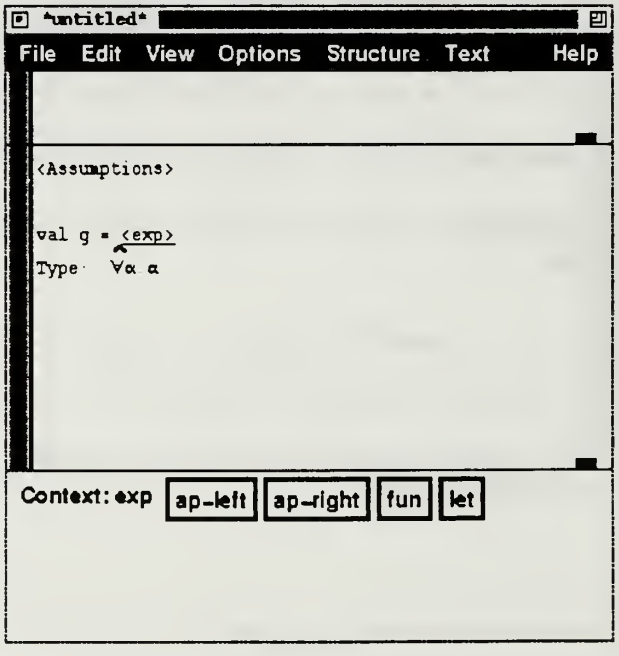

Figure 5.10: The view when the definition is named g.

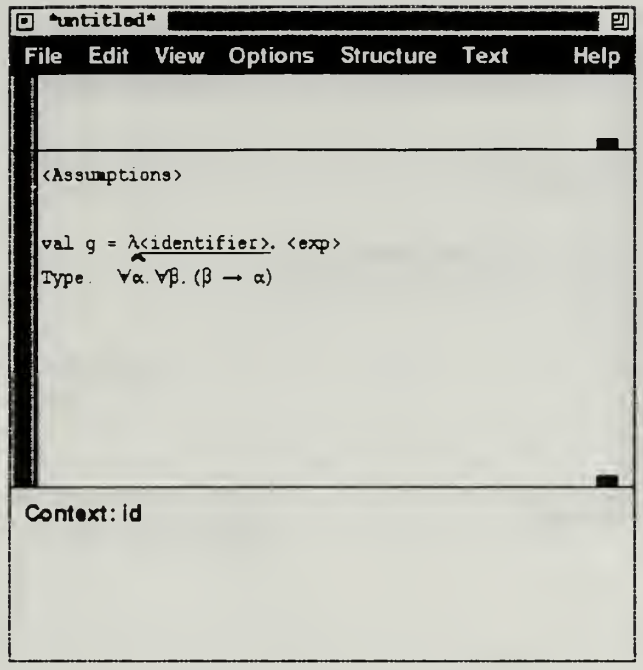

Figure 5.11: The context **fun** has been selected and the place holders for a λ expression have been inserted.

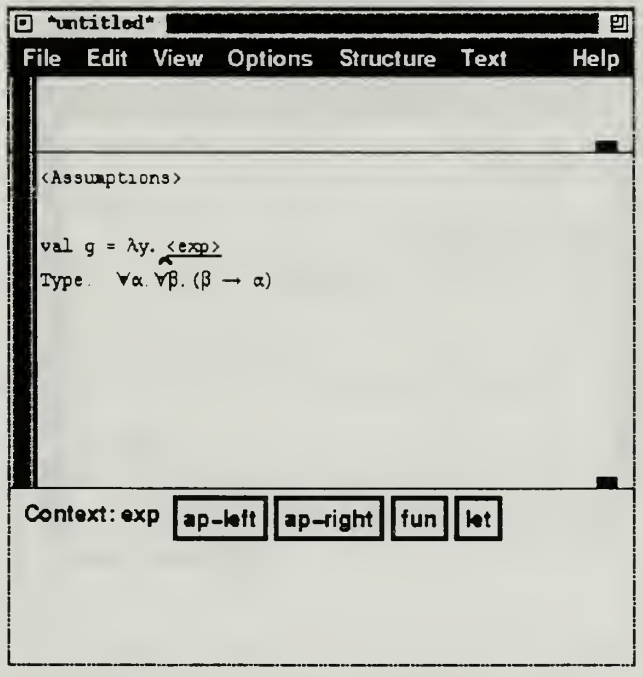

Figure 5.12: The identifier for the λ abstraction has been entered and the context has been moved to the expression.

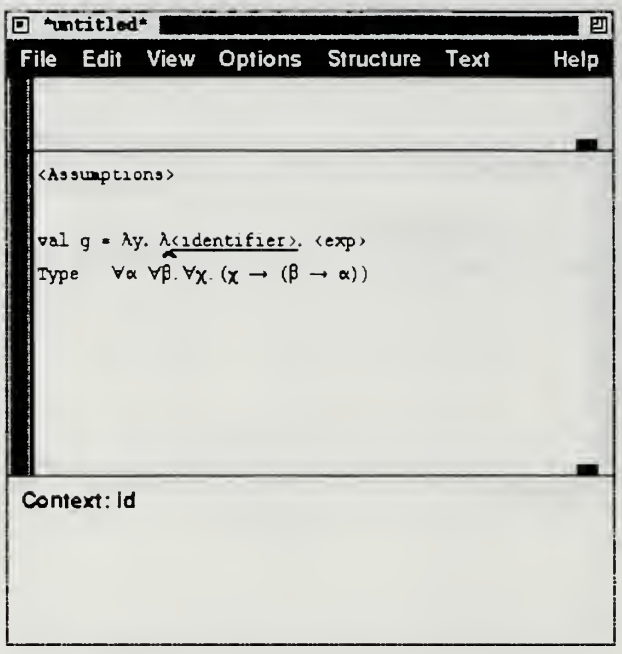

Figure 5.13: A second λ abstraction is entered.

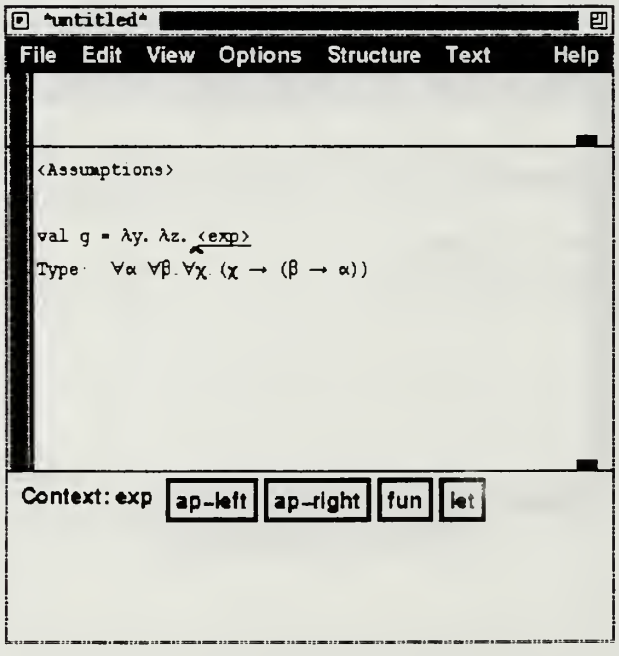

Figure 5.14: The identifier for the second λ abstraction has been entered and the context has been moved to the expression.

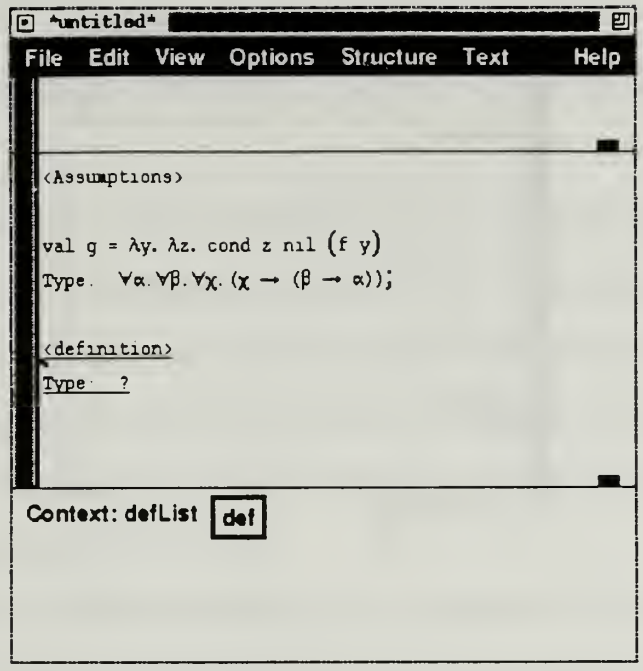

Figure 5.15: The rest of the definition *g* has been entered.

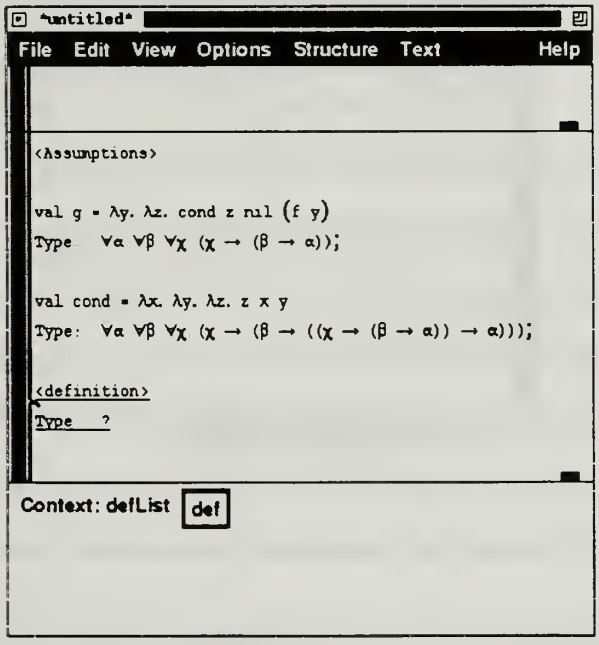

Figure 5.16: The definition for the conditional *cond* has been entered.

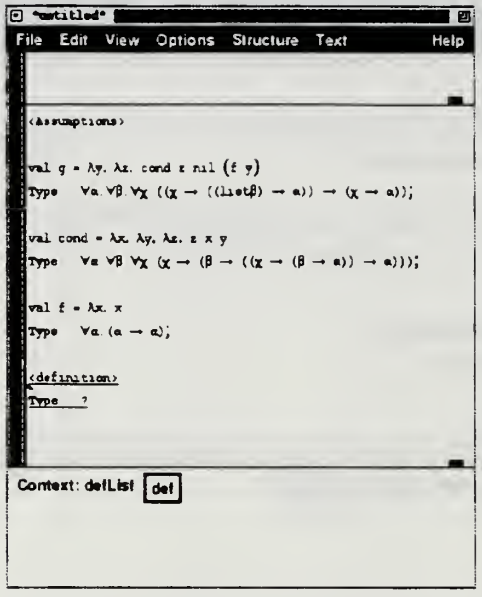

Figure 5.17: A definition for f has been entered and g's type has changed.

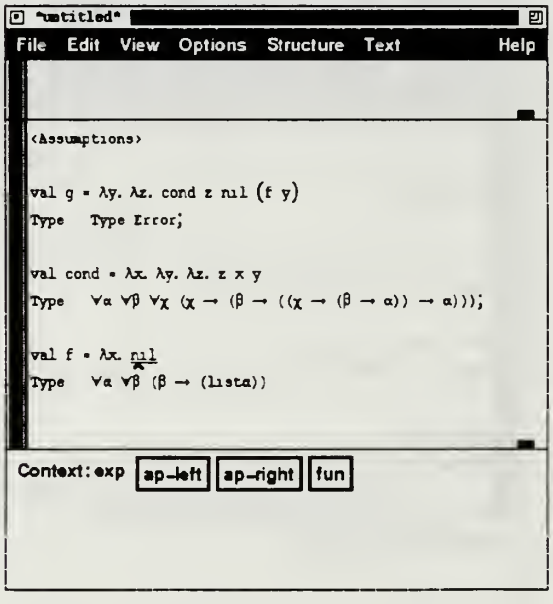

Figure 5.18: The definition for f has been changed causing g to be retyped resulting in a type error for g.

VI. RELATED WORK AND CONCLUSIONS

We have shown that on-line type inference, an essential element of any interactive programming environment, is possible using an attribute grammar. Our type system obtains typings consistent with the Hindley-Milner type system. Type checking is performed on-line and incrementally. Modular, top-down program development is possible and the type system maintains correct typings for each definition at every stage of program development.

The on-line type inference problem can be reduced to that of first order unification on a set of type equations. We use the well known reduction from the Hindley-Milner type system to first order unification. The reduction generates type equations for each definition. The type equations can be saved and used in future typings for the definition which would then just require reunifying the type equations. The type equations will not change unless the expression is modified (by editing), at which point the type equations would be reconstructed.

The algorithm is incremental because we can save the type equations and only reunify the type equations when we must reinfer the type for a definition. Additionally, the attribute grammar model we propose has a certain degree of incremental attribute re-evaluation implicit in the model. Significant additional savings are possible if the dependencies of the definitions are observed. The dependency relation is a partial order and thus we can generate a total (linear) order. For a modified definition we must retype only the modified definition and those definitions that depend on the modified definition.

A. RELATED WORK

MIT's Id programming environment is the current state of the art. Our type system is more incremental because we take advantage of the reduction from the Hindley-Milner type system to first-order unification. Any time a definition's type must be reinferred we only have to reunify its type equations. The *Id* environment calls *W* from the Hindley-Milner type system which does all the work of constructing and unifying the type equations for an expression every time it is called.

B. FUTURE WORK

This thesis will serve as the basis for further research aimed at ultimately developing a type discipline for a class of implicitly type imperative programming languages in an interactive programming environment.

One issue we have not addressed in this paper is what happens if more than one definition is entered with the same name. In a syntax directed editor environment that we have proposed, the ability to edit any definition makes the process of updating a sequence of definitions occur in a framework that is more intuitive than current environments i.e., Prolog, Standard ML, and Scheme. What we have not considered, however, is what is the meaning of two definitions in a sequence with the same name. This is called overloading. Overloading is an independent problem that has been studied separately [Ref. 3]. Overloading is beyond the scope of this paper, so the effect of multiple definitions with the same name has not been addressed in this thesis. Integrating the on-line system with overloading remains to be done.

Investigating Sagiv et al.'s CAG to NCAG transformation is another area where further research is needed. Since *SynGen* is only capable of handling non-circular attribute equations the simple attribution in Chapter IV can not be implemented using *SynGen*. Work remains in determining whether the transformation of [Ref. 12]

can be applied to transform the circular attribution of Chapter IV, Another interesting research direction is exploring whether a fixed-point finding attribute evaluator can be produced automatically using the work of Farrow [Ref. 5].

LIST OF REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1988.

[2] Anthony Field and Peter Harrison. *Functional Programming.* Addison Wesley, 1987.

[3] Bruce J. Bull. Type inference with overloading using an attribute grammar. Master's thesis, Naval Postgraduate School, Monterey CA, 1994.

[4] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[5] R. W. Farrow. Automatic generation of fixed-point-finding evaluators for circular but well-defined attribute grammars. *Symposium on Compiler Construction*, pages 85–98, 1986.

[6] GrammaTech Inc., Ithaca, NY. *The Synthesizer Generator Reference Manual, Release 4.0*, 1992.

[7] U. Kastens. Ordered attribute grammars. *Acta Inf*, 13,3:229–256, 1980.

[8] Luis Damas and Robin Milner. Principal type schemes for functional programs. *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[9] Alan Mycroft. Incremental polymorphic type checking with update (preliminary version). *Proc. Symposium on Logical Foundations of Computer Science, LNCS*, 620:347–357, 1992.

[10] Rishiyur S. Nikhil. Practical polymorphism. *Proc. Conf. on Functional Programming Languages and Computer Architecture.*, LNCS 201:319–333, 1985.

[11] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[12] S. Sagiv, O. Edelstein, N. Francez, and M. Rodeh. Resolving circularity in attribute grammars with applications to data flow analysis (preliminary-version). *Association for Computing Machinery*, 36, 1989.

[13] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. *Prof. 5th Conf. on Functional Programming Languages and Computer Architecture, LNCS*, 523:379–405, 1991.

[14] Mitchell Wand and Patrick O'Keefe. On the complexity of type inference with coercion. *Proc. ACM Conf. on Functional Programming and Computer Architecture*, 1989.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304–6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943–5002	2
3.	Chairman, Computer Science Department Code CS Naval Postgraduate School Monterey, CA 93943	2
4.	Dr. Dennis M. Volpano Code CS/Vo Naval Postgraduate School Monterey, CA 93943	10
5.	Dr. Timothy Shimeall Code CS/Sm Naval Postgraduate School Monterey, CA 93943	1
6.	LT Thomas L. Robinson 200 Main St. Kingston, MA 02364	2